Towards a Hybrid Verification Methodology for Communication Protocols (Short Paper)

Christian Bartolo Burlò[1(✉)], Adrian Francalanza[1(✉)], and Alceste Scalas[2(✉)]

[1] University of Malta, Msida, Malta
{christian.bartolo.16,adrian.francalanza}@um.edu.mt
[2] Aston University, Birmingham, UK
a.scalas@aston.ac.uk

Abstract. We present our preliminary work towards a comprehensive solution for the hybrid (static + dynamic) verification of open distributed systems, using session types. We automate a solution for binary sessions where one endpoint is statically checked, and the other endpoint is dynamically checked by a *monitor* acting as an intermediary between typed and untyped components. We outline our theory, and illustrate a tool that *automatically synthesises* type-checked session monitors, based on the Scala language and its session programming library (`lchannels`).

Keywords: Session types · Static and dynamic verification · Monitors

1 Introduction

Session Types [12,13,27] have emerged as a central formalism for the verification of concurrent and distributed programs. Session-types-based analysis ensures that a program correctly implements some predetermined *communication protocol*, stipulating the desired exchange of messages [4,16]. The analysis is typically performed *statically*, via type checking, before the programs are deployed. However, full static analysis is not always possible (*e.g.,* when the source code of third-party programs and components is unavailable); in such cases, session types are checked at runtime via *monitors* [6,10,17,19]. We view these approaches as two extremes on a continuum: our aim is to develop practical *hybrid* (static and dynamic) verification methodologies and tools for distributed programs in *open* settings. In particular, our aim is to verify distributed systems where:

The research was partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233.

Published by Springer Nature Switzerland AG 2020
A. Gotsman and A. Sokolova (Eds.): FORTE 2020, LNCS 12136, pp. 227–235, 2020.
https://doi.org/10.1007/978-3-030-50086-3_13

(i) we make no assumptions on how messages are delivered between components;
(ii) the components available prior-deployment are checked statically; and
(iii) the components that are unavailable for checking prior-deployment are verified at runtime, by deploying *autogenerated, type-checked monitors.*

To achieve this aim, we present a methodology with three key features, presented as contributions in this paper:

F1. Open systems are prone to malicious attacks and data corruption. Thus, we describe protocols via *augmented session types* including *runtime data assertions* (reminiscent of interaction refinements [18]), and synthesise *monitors* that automate such data checks. Unlike [6,18], our monitors are independent, type-checked processes, that can be deployed over any network.
F2. We develop a tool that, given a session type S, can synthesise the Scala code of *(1)* a *type-checked monitor* that verifies at run-time whether an interaction abides by S (aim *(iii)*), and *(2)* the *signatures* usable to implement a process that interacts according to S, in a correct-by-construction manner (aim *(ii)*).
F3. Our monitor synthesis can *abstract over low-level communication protocols*, bridging across a variety of message transports (*e.g.,* TCP/IP, REST, *etc.*): this is key to facilitate the interaction with third-party (untyped) components in open systems (aim *(i)*); this is also unlike previous work on session monitoring (theoretical [6] or practical [19]) that focus on a specific technology and runtime system, or assume a centralised message routing medium.

2 Binary Sessions with Assertions

A session type defines the intended behaviour of a participant that communicates with another over a *channel.* Our work is based on *session types with assertions*:

$$\begin{array}{rrcl}
\text{Assertions} & A & ::= & v_1 == v_2 \mid v_1 >= v_2 \mid A_1 \mathbin{\&\&} A_2 \mid !A \mid \dots \\
\text{Base types} & B & ::= & \mathsf{Int} \mid \mathsf{Str} \mid \mathsf{Boolean} \mid \dots \\
\text{Session types} & R, S & ::= & \&_{i\in I}?\mathtt{l}_i(V_i : B_i)[A_i].S_i \mid \oplus_{i\in I}!\mathtt{l}_i(V_i : B_i)[A_i].S_i \\
 & & & \mid \mathsf{rec}\ X.S \mid X \mid \mathsf{end} \quad \text{(with } I \neq \emptyset,\ \mathtt{l}_i \text{ pairwise distinct)}
\end{array}$$

We assume a set of *base types* B, and introduce *payload identifiers* V (with their types) and *assertions* A (*i.e.*, predicates on payload values). *Branching* (or *external choice*) $\&_{i\in I}?\mathtt{l}_i(V_i : B_i)[A_i].S_i$ requires the participant to receive one message of the form $\mathtt{l}_i(v_i)$, where v_i is of (base) type B_i for some $i \in I$; the value v_i (i.e., the message payload) is bound to the variable V_i in the continuation. If the assertion $A_i\,[^{v_i}/_{V_i}]$ holds, the participant must proceed according to the *continuation type* $S_i\,[^{v_i}/_{V_i}]$, but if the assertion fails, a violation is raised. *Selection* (or *internal choice*) $\oplus_{i\in I}!\mathtt{l}_i(V_i : B_i)[A_i].S_i$ requires the participant to choose and send one message $\mathtt{l}_i(v_i)$ where v_i is of (base) type B_i for some $i \in I$; a violation is raised if the assertion $A_i\,[^{v_i}/_{V_i}]$ does *not* hold, otherwise the protocol proceeds as $S_i\,[^{v_i}/_{V_i}]$. The *recursive* session type $\mathsf{rec}\ X.S$ binds the recursion variable X

in S (we assume guarded recursion), while end is the *terminated* session. For brevity, we often omit $\oplus$ and $\&$ for singleton choices, end, and trivial assertions (*i.e.*, true). A process implementing a session type S can correctly interact with a process implementing the *dual type of* S, denoted $\overline{S}$—where each selection (resp. branching) of S is a branching (resp. selection), with the same choices:

$$\overline{\&_{i\in I}?\mathtt{l}_i(V_i : B_i)[A_i].S_i} \;=\; \oplus_{i\in I}!\mathtt{l}_i(V_i : B_i)[A_i].\overline{S_i} \qquad \overline{\mathsf{end}} \;=\; \mathsf{end} \qquad \overline{X} \;=\; X$$
$$\overline{\oplus_{i\in I}!\mathtt{l}_i(V_i : B_i)[A_i].S_i} \;=\; \&_{i\in I}?\mathtt{l}_i(V_i : B_i)[A_i].\overline{S_i} \qquad \overline{\mathsf{rec}\ X.S} \;=\; \mathsf{rec}\ X.\overline{S}$$

Example 1. The type S_{login} below describes the protocol of a server handling *authorised logins*. Notice that the type uses two assertion predicates:

- *validAuth()* checks if an OAuth2-style token [20] authorises a given user;
- *validId()* checks whether an authentication id is correct for a given user.

$$\begin{aligned} S_{login} \;=\; & \mathsf{rec}\ X.?\mathtt{Login}(uname{:}\mathsf{Str}, pwd{:}\mathsf{Str}, tok{:}\mathsf{Str})[validAuth(uname, tok)]. \\ & \oplus\left\{!\mathtt{Success}(id{:}\mathsf{Str})[validId(id, uname)].R\ ,\ !\mathtt{Retry}().X\right\} \end{aligned}$$

The server waits to receive $\mathtt{Login}(uname{:}\mathsf{Str}, pwd{:}\mathsf{Str}, tok{:}\mathsf{Str})$, where *tok* is a token obtained by the client from an authorisation service. Once received, the values of *uname* and *tok* are passed to the predicate *validAuth*() which checks whether *tok* contains a desired cryptographically-signed authorisation for *uname*: if it evaluates to true, the server can either send `Success` including an *id*, or `Retry`. If the server chooses the former, then *id* and *uname* must be validated by *validId*(): if it succeeds, the message is sent and the server continues along session type R. If the server chooses to send `Retry`, the session loops. ■

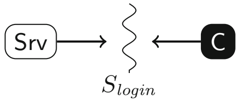
3 Design and Implementation

We now give an overview (Sect. 3.2) and an example-driven tour (Sect. 3.3) of our methodology; but first, we summarise the toolkit underlying its implementation (Sect. 3.1).

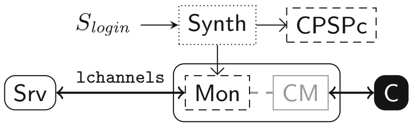
3.1 Background: Session Programming with `lchannels`

`lchannels` [21,24,25] is a library implementation of session types in the Scala programming language. Its API is inspired by the continuation-passing encoding of session types into the linear π-calculus [9]. `lchannels` allows to implement a program that communicates according to a session type S by *(1)* translating S into a set of *Continuation-Passing Style Protocol classes* (CPSPc), capturing the order of send/receive operations in S; and *(2)* communicating via "one-shot" channel objects, having type `Out[A]` or `In[A]`—where `A` is a CPSP class. We show an example of CPSPc in Sect. 3.3. The main payoffs of `lchannels` are that *(1)* the CPSP classes restrict the usage the `In[A]`/`Out[A]` channel objects to receive/send messages according to S, letting the Scala compiler check *safety* (*i.e.*, only messages allowed by S are sent) and *exhaustiveness* (*i.e.*, all inputs allowed by S are handled); and *(2)* the library provides run-time linearity enforcement: *e.g.,* if a “one-shot” channel object is used twice to send messages, then the program is not advancing along S, hence `lchannels` discards the message and raises an error.

3.2 Hybrid Verification via Static and Dynamic Checking



We now illustrate how our methodology is implemented, as a tool [7] targeting the Scala programming language. Consider the scenario on the right: a client C exchanges messages with a server Srv over a network. Srv implements the session type S_{login} outlined in Example 1, and expects each client C to follow the dual, $\overline{S_{login}}$. However, in an open system, we cannot guarantee that C abides by $\overline{S_{login}}$.



Our approach is outlined on the left. The accessible participant Srv is *statically* checked, and the behaviour of the inaccessible participant C is *dynamically* monitored at runtime. Given S_{login}, the synthesiser Synth generates *(1)* the *Continuation-Passing-Style Protocol classes* (CPSPc) for representing S_{login} in Scala and `lchannels` (see Sect. 3.1), and *(2)* the source code of a runtime monitor (Mon), based on the CPSPc above. Below are the CPSPc generated from S_{login} by our synthesiser: notably, they can be used to write a type-checked version of Srv.

```scala
case class Login(uname: String, pwd: String, tok: String)(val cont: Out[Choice1])
sealed abstract class Choice1
case class Success(id: String)(val cont: Out[R]) extends Choice1
case class Retry()(val cont: Out[Choice1]) extends Choice1
case class R(...) // This is the continuation of the session (omitted)
```

The messages sent from Srv to C (and *vice versa*) must pass through the monitor Mon. As Srv and Mon use `lchannels` to interact, they are statically typed according to S_{login} and $\overline{S_{login}}$; instead, there is no assumption on the interaction between Mon and C: it is handled by a user-supplied *connection manager* (CM), which acts as a *translator* and *gatekeeper* by transforming messages from the transport protocol supported by C to the Mon’s CPSP classes, and *vice versa.* Hence, CM provides a message transport abstraction for Mon and Srv: to support new clients and message transports, only CM needs extending.

When the monitor is initialised, it invokes CM to set up the communication channel with client C, through a suitable message transport: *e.g.*, in the case of TCP/IP, CM creates a socket and initialises the I/O buffers. Each message sent from Srv to Mon via `lchannels` is analysed by Mon, and if it conforms to S_{login} and its assertions, it is translated by CM and forwarded to C. Dually, each message sent from C to Mon is translated by CM and analysed by Mon, and if it conforms to $\overline{S_{login}}$ and its assertions, it is forwarded to Srv. Mon’s assertion checks provide additional verification against incorrect values from Srv or C.

3.3 A Step-by-Step Example

To illustrate our approach and implementation, we now follow the message exchanges prescribed by S_{login}, showing how they engage with the elements of our design. Roughly, Mon acts as a state machine: it transitions by receiving and forwarding messages between Srv and CM, abiding by the type S_{login} and its dual. CM, in turn, provides a `send`/`receive` interface to Mon, and delivers messages to/from client C. The monitor also maintains a mapping, called `payloads`, that associates the payload identifiers of S_{login} to their current values.

```
1  val loginR = """LOGIN (.+) (.+) (.+)""".r
2  def receive(): Any = inBuf.readLine() match {
3    case loginR(uname, pwd, tok) => Login(uname, pwd, tok)(null)
4    case other => other
5  }
```

We begin with the login request sent from a client over TCP/IP. The client's message is initially handled by the connection manager CM, which provides a `receive` method like the one shown above: it is invoked by Mon to retrieve messages. When invoked, `receive` checks the socket input buffer `inBuf`: if a new supported message is found (line 3, where the message matches the regex `loginR`), the corresponding CPSP class is returned to the monitor; otherwise, the unaltered message is returned (line 4).

```
1  def receiveLogin(srv: Out[Login], client: ConnManager): Unit = {
2    client.receive() match {
3      case msg @ Login(_, _, _) =>
4        if (validateAuth(msg.uname, msg.tok)) {
5          val cont = srv !! Login(msg.uname, msg.pwd, msg.tok)_
6          payloads.Login.uname = msg.uname
7          sendChoice1(msg.cont, client) // Protocol continues
8        } else { /* log and halt: Incorrect values received */ }
9      case _ => /* log and halt: Unexpected message received */
10 } }
```

On the left is the synthesised code for Mon that handles the beginning of S_{login}. The monitor invokes CM's method `receive` (shown above) to retrieve the latest message (line 2). Depending on the type of message, the monitor will perform a series of actions. By default, a catch-all case (line 9) handles any messages violating the protocol. If `Login` is received, the monitor initially invokes the function `validateAuth()` with the values of `uname` and `tok`; *i.e.*, the assertion predicate in S_{login} corresponds to a Scala function (imported from a user-supplied library). If the function returns `true`, the message is forwarded to the server Srv (line 5), otherwise the monitor logs the violation and halts. The function used to forward the message (`!!`), which is part of `lchannels`, returns a continuation channel that is stored in `cont`. The value of `uname` is stored in a mapping (line 6) since it is used later on in S_{login}. Finally, the monitor moves to the next state, by calling the synthesised method `sendChoice1`, passing `cont` to continue the protocol.

```
1  def sendChoice1(srv: In[Choice1], Client: ConnManager): Unit = {
2    srv ? {
3      case msg @ Success(_) =>
4        if (validateId(msg.id, payloads.Login.uname)) {
5          Client.send(msg)
6          /* Continue according to R */
7        } else {
8          /* log and halt: Sending incorrect values. */
9        }
10     case msg @ Retry() =>
11       Client.send(msg)
12       receiveLogin(msg.cont, Client)
13 } }
```

On the left is the synthesised code of Mon that handles the server's response to the client. According to S_{login}, the server can choose to send either `Success` or `Retry`; correspondingly, the monitor waits to receive either of the options from Srv, using the function `?` from `lchannels` (line 2).

- If the server sends `Success`, including the value *id* as specified in S_{login}, the first case is selected (line 3). The monitor evaluates the assertion on *id* and *uname* (stored in `receiveLogin` above, and now retrieved from the `payloads` mapping): if it is satisfied, the message is sent to the client (line 5) via CM's `send` method (explained below), and the monitor continues according to session type R. Otherwise, the monitor logs a violation and halts (line 8).
- Instead, if the server sends `Retry` (line 10), the message is forwarded directly to the client using the method `send` of the CM (see below); notice that there are no dynamic checks at this point, as there is no assertion after *Retry* in S_{login}. The monitor then goes back to the previous state `receiveLogin`.

Notably, unlike the synthesised code of `receiveLogin` (that handles the previous external choice), there is no catch-all case for unexpected messages from Srv. In fact, here we assume that Srv is written in Scala and `lchannels`, hence statically checked, and conforming to S_{login}; hence, it can only send one of the expected messages (as per Sect. 3.1). The monitor only checks the assertions on Srv's messages.

```
1 def send(msg: Any): Unit = msg match {
2   case Success(id) => outB.write(f"SUCCESS ${id}\n")
3   case Retry() => outB.write(f"RETRY\n")
4   case _ => { close();
5               throw new Exception("Invalid message") }
6 }
```

Finally, we review the `send` method of CM: it translates messages from a CPSP class instance to the format accepted by the client's transport protocol. In this case, the format is a textual representation of the session type. The catch-all case (lines 4–5) is for debugging purposes.

4 Conclusion

We presented our preliminary work on the hybrid verification of open distributed systems, based on *session types with assertions* and *automatically syntehsised monitors*—with a supporting tool [7] based on the Scala programming language.

Future Work. Our approach adheres to the *"fail-fast"* design methodology: if an assertion fails, the monitor logs the violation and halts. In the practice of distributed systems, “fail-fast” is advocated as an alternative to defensive programming [8]; it is also in line with existing literature on runtime verification [5]. Our further rationale for this design choice is that we intend to investigate *monitorability* properties of session types, along the lines of recent work [1,2,11], and identify any limits, in terms of what can be verified at runtime. We plan to extend our approach to multiparty sessions [14,15], in connection to existing work [22,23] based on `lchannels` and Scribble [26,28]. Finally, we plan to investigate how to handle assertion violations by adding *compensations* to our session types, formalising how the protocol should proceed whenever an assertion fails.

Related Work. The work in [6] formalises a theory of monitored (multiparty) session types, based on a global, centralised router providing a *safe transport network* that dispatches messages between participant processes. The main commonality with our work is that session types are used to synthesise monitors. The main differences (besides our focus on a tool implementation) are that *(1)* we do not assume a centralised message routing system, and consider the network adversarial (as per contribution F1) and use monitors to also protect typed participants; *(2)* our monitors can enforce data constraints, through assertions; and *(3)* in our setting, if a participant sends an invalid message, the monitor will flag violations (and stop computation) whereas [6] drops the invalid message, but will continue forwarding the rest, akin to runtime enforcement via suppressions [3]. Our protocol assertions are reminiscent of *interaction refinements* in [18], that are also statically generated (by an F# type provider), and dynamically enforced when messages are sent/received. However, we enforce our assertions by synthesising well-typed monitoring processes that can be deployed over a network, whereas [18] injects dynamic checks in the local executable of a process.

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. PACMPL **3**(POPL), 1–29 (2019). https://doi.org/10.1145/3290365
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: An operational guide to monitorability. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 433–453. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30446-1_23
3. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On runtime enforcement via suppressions. In: CONCUR, LIPIcs, vol. 118. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018). https://doi.org/10.4230/LIPIcs.CONCUR.2018.34
4. Ancona, D., et al.: Behavioral types in programming languages. Found. Trends Program. Lang. **3**(2–3), 95–230 (2016). https://doi.org/10.1561/2500000031
5. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1

6. Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In: Beyer, D., Boreale, M. (eds.) FMOODS/FORTE - 2013. LNCS, vol. 7892, pp. 50–65. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_5
7. Burlò, C.B., Francalanza, A., Scalas, A.: STMonitor implementation (February 2020). https://github.com/chrisbartoloburlo/stmonitor
8. Cesarini, F., Thompson, S.: ERLANG Programming, 1st edn. O'Reilly Media Inc., Sebastopol (2009)
9. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/j.ic.2017.06.002
10. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python. Form. Methods Syst. Des. **46**(3), 197–225 (2014). https://doi.org/10.1007/s10703-014-0218-8
11. Francalanza, A., Aceto, L., Ingolfsdottir, A.: Monitorability for the Hennessy–Milner logic with recursion. Form. Methods Syst. Des. **51**(1), 87–116 (2017). https://doi.org/10.1007/s10703-017-0273-z
12. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_35
13. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
14. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL. ACM (2008). https://doi.org/10.1145/1328438.1328472. Full version in [15]
15. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 1–67 (2016). https://doi.org/10.1145/2827695
16. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 1–36 (2016). https://doi.org/10.1145/2873052
17. Jia, L., Gommerstadt, H., Pfenning, F.: Monitors and blame assignment for higher-order session types. In: POPL. ACM (2016). https://doi.org/10.1145/2837614.2837662
18. Neykova, R., Hu, R., Yoshida, N., Abdeljallal, F.: A session type provider: Compile-time API generation of distributed protocols with refinements in f#. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018. ACM (2018). https://doi.org/10.1145/3178372.3179495
19. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: Proceedings of the 26th International Conference on Compiler Construction. ACM (2017). https://doi.org/10.1145/3033019.3033031
20. OAuth Working Group: RFC 6749: OAuth 2.0 framework (2012). http://tools.ietf.org/html/rfc6749
21. Scalas, A.: lchannels (2017). https://alcestes.github.io/lchannels
22. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming. In: ECOOP, Leibniz International Proceedings in Informatics (LIPIcs), vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.ECOOP.2017.24
23. Scalas, A., Dardha, O., Hu, R., Yoshida, N.: A linear decomposition of multiparty sessions for safe distributed programming (artifact). DARTS **3**(2) (2017). https://doi.org/10.4230/DARTS.3.2.3

24. Scalas, A., Yoshida, N.: Lightweight session programming in scala. In: ECOOP, LIPIcs, vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.ECOOP.2016.21
25. Scalas, A., Yoshida, N.: Lightweight session programming in scala (artifact). DARTS **2**(1) (2016). https://doi.org/10.4230/DARTS.2.1.11
26. Scribble Homepage (2020). http://www.scribble.org
27. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Maritsas, D., Philokyprou, G., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58184-7_118
28. Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch Lafuente, A. (eds.) TGC 2013. LNCS, vol. 8358, pp. 22–41. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05119-2_3